2

Technical Report 1429

Maygen: A Symbolic Debugger Generation System

Christine L. Tsien

MIT Artificial Intelligence Laboratory

KEPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY *(Leave blank)*	**2. REPORT DATE** July 1993	**3. REPORT TYPE AND DATES COVERED** technical report

4. TITLE AND SUBTITLE
Maygen: A Symbolic Debugger Generation System

5. FUNDING NUMBERS
N00014-91-J-4038

6. AUTHOR(S)
Chrstine L. Tsien

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Artificial Intelligence Laboratory
Massachusetts Institute of Technology
545 Technology Square
Cambridge, Massachusetts 02139

8. PERFORMING ORGANIZATION REPORT NUMBER
AI-TR 1429

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)
Office of Naval Research
Information systems
Arlington, Virginia 22217

10. SPONSORING/MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES
None

12a. DISTRIBUTION/AVAILABILITY STATEMENT
Distribution of this document is unlimited

12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*

With the development of high-level languages for new computer architectures comes the need for appropriate debugging tools as well. One method for meeting this need would be to develop, from scratch, a symbolic debugger with the introduction of each new language implementation for any given architecture. This, however, seems to require unnecessary duplication of effort among developers. Compilation technology has alleviated some duplication of effort in the development of compilers. Can similar ideas aid in the efficient development of symbolic debuggers as well?

Maygen explores the possibility of making debugger development efficient by influencing the language and architecture development processes. Maygen is a "debugger generation

(continued on back)

14. SUBJECT TERMS

debugging	generation
symbolic debugging	multilingual debugging

15. NUMBER OF PAGES
85

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	**18. SECURITY CLASSIFICATION OF THIS PAGE**	**19. SECURITY CLASSIFICATION OF ABSTRACT**	**20. LIMITATION OF ABSTRACT**
UNCLASSIFIED	UNCLASSIFIED	UNCLASSIFIED	UNCLASSIFIED

Block 13 continued:

system," built upon the idea that symbolic debuggers can be divided into three components: a set of source language interface routines, a set of machine architecture interface routines, and a language-independent and architecture-independent debugger skeleton. Maygen then exploits this modularity: First, Maygen precisely defines as well as houses the language-independent and architecture-independent debugger skeleton. Second, Maygen defines the protocol for interface interaction among source language developers, machine architecture developers, and the general-purpose debugger skeleton. Finally, Maygen provides a framework in which the resident debugger skeleton is automatically developed into a stand-alone symbolic debugger; the resulting debugger is tailored to the specific provisions of a particular language group and a particular architecture group.

Maygen: A Symbolic Debugger Generation System

by

Christine L. Tsien

S.B. Computer Science and Engineering, Massachusetts Institute of Technology (1991)

Submitted to the Department of Electrical Engineering and Computer Science in partial fulfillment of the requirements for the degree of

Master of Science in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

May 1993

Maygen: A Symbolic Debugger Generation System

by

Christine L. Tsien

Submitted to the Department of Electrical Engineering and Computer Science
on May 7, 1993, in partial fulfillment of the
requirements for the degree of
Master of Science in Electrical Engineering and Computer Science

Abstract

With the development of high-level languages for new computer architectures comes the need for appropriate debugging tools as well. One method for meeting this need would be to develop, from scratch, a symbolic debugger with the introduction of each new language implementation for any given architecture. This, however, seems to require unnecessary duplication of effort among developers. Compilation technology has alleviated some duplication of effort in the development of compilers. Can similar ideas aid in the efficient development of symbolic debuggers as well?

Maygen explores the possibility of making debugger development efficient by influencing the language and architecture development processes. Maygen is a "debugger generation system," built upon the idea that symbolic debuggers can be divided into three components: a set of source language interface routines, a set of machine architecture interface routines, and a language-independent and architecture-independent debugger skeleton. Maygen then exploits this modularity: First, Maygen precisely defines as well as houses the language-independent and architecture-independent debugger skeleton. Second, Maygen defines the protocol for interface interaction among source language developers, machine architecture developers, and the general-purpose debugger skeleton. Finally, Maygen provides a framework in which the resident debugger skeleton is automatically developed into a stand-alone symbolic debugger; the resulting debugger is tailored to the specific provisions of a particular language group and a particular architecture group.

Thesis Supervisor: Thomas F. Knight, Jr., Ph.D.
Title: Principal Research Scientist, Department of Electrical Engineering and Computer Science

Thesis Supervisor: Alan L. Davis, Ph.D.
Title: Company Supervisor, Hewlett Packard Laboratories

To my parents

Acknowledgments

First, I would like to thank Al Davis for being a great leader and advisor, for understanding that with high morale and interesting work naturally follows true motivation and quality performance. (I.e., occasional goof-off days, such as group outings to see Terminator 2 or watch the Giants, keeps people happy and diligent through subsequent crunch times.) I would also like to thank him for his careful reading of my thesis draft. I wish him all the best with Mayfly as well as his future endeavors.

I would like to thank Tom Knight for agreeing to be my MIT supervisor, for being positive and supportive of my work even though the scope or focus seemed to change nearly every time I flew to MIT for a meeting, and for being interested in everything, thus making him a great resource for a diverse set of questions.

I would like to thank Mike Lemon for his unwavering support and friendship since my first HP summer in 1989. More recently, I am indebted to him for letting me clutter half of his disk space with my backups, for his eloquent exposition of abstract machines during one of my periods of confusion, and for subsequently letting me borrow heavily from that description for my introductory paragraphs of Section 4.2.

I would like to thank Robin Hodgson for helping to flesh out the preliminary debugger generation idea, for helping me to understand the Mayfly, and for having done a lot of work on Maydebug, the guts of which went into much of my Mayfly test case. I also want to thank the Mayfly group overall for providing a very enjoyable work environment.

Next, thanks go to John Conery for explaining the OPAL system and for having done a lot of OPAL/OM work, the guts of which went into much of my OPAL and OM test cases.

I would like to thank Bill Dally for being supportive of my medical interests and especially for signing my registration even when he thought I was taking too many classes.

I would like to thank all of the MEDG members for being my foster group at MIT while I was finishing Maygen work and for listening to my thesis talk; the talk format contributed greatly to my subsequent decision on how to structure my written thesis presentation.

Of course I also want to thank all of my friends—not only those at MIT, who gave me much needed and relaxing breaks from work, but also those who have left MIT but still remain close to me in spirit and in email.

Special thanks go to Jamie: in all of my busiest times, he alone was still able to convince me to take three to five mini-breaks a day. (It was either that, or spend twice as much time cleaning up doggy accidents!)

I feel compelled to thank the Association of American Medical Colleges for scheduling the MCATs to be three weeks before the thesis deadline; had the MCATs been at a different time, I might not have had as good an excuse for not studying as much as one should.

As always, I am very thankful to my parents and my sister for their continual support, guidance, patience, interest, and enthusiasm in all of my endeavors.

I wholeheartedly thank God for helping me with all that I do, as well as for allowing my biggest problem to be having too many choices (along with a flair for indecisiveness).

Last, but definitely not least, I would like to thank Brad Spiers for all of his love, friendship, laughter, support, and encouragement. I thank him for not lifting an eyebrow at my cutting coupons and reading grocery store ads in the midst of MCAT studies and thesis work. I thank him for correcting my almost-clichés and colloquialisms; if it weren't for him, I'd be flushing out ideas and saying, "Close, but no banana." Finally, I want to thank him for letting me sign him up for Columbia House Video Club membership (and thus getting those ten great new movies for a low price!) just when we most needed to work. :]

This paper describes research conducted at Hewlett Packard Laboratories, as part of the MIT VI-A Internship Program, and at the Artificial Intelligence Laboratory of the Massachusetts Institute of Technology. Support for this research was provided in part by a National Science Foundation Graduate Research Fellowship. Support for the laboratory's artificial intelligence research is provided in part by the Advanced Research Projects Agency of the Department of Defense under Office of Naval Research contract N00014-91-J-4038.

About the Author

Christine Tsien was born on the 28th of November, 1969, in Minneapolis, Minnesota. She was educated in public schools, graduating valedictorian from Mounds View High School in Arden Hills, Minnesota, in 1987. With the financial aid of a National Merit Scholarship, she was able to attend the Massachusetts Institute of Technology, where she majored in computer science, concentrated in Russian language, and maintained an interest in biology and medicine. As a sophomore, she was invited to participate in the VI-A program with Hewlett Packard Laboratories in Palo Alto, California. She was elected to and is a member of Tau Beta Pi, Eta Kappa Nu, and Sigma Xi, and she served as Vice President and Social Chair for Eta Kappa Nu during the 1990-91 academic year. She has been a member of the Society of Women Engineers for six years, during which she served on the Executive Committee and the Financial Committee for one year each and as Treasurer for two years. She is also a member of the Association for Computing Machinery and the Biomedical Engineering Society. She earned her Bachelor of Science degree from the Department of Electrical Engineering and Computer Science in June, 1991. The author was accepted into the doctoral program at the Massachusetts Institute of Technology, where she recently finished her Master of Science degree with the financial support of a National Science Foundation Graduate Fellowship.

During her years at the Massachusetts Institute of Technology, the author also participated in Alpha Phi Omega National Service Fraternity, Figure Skating Club, Freshman Associate Advising, Tech Square Big Sisters, Chinese Students' Club, and Project Contact. She was a laboratory teaching assistant in the Biology Department, engaged in research at the Laboratory for Computer Science and at the Artificial Intelligence Laboratory, and worked various jobs in West Campus Houses, ARA Food Service, and Hayden Library. She also volunteered at Mount Auburn, Boston City, and Massachusetts General Hospitals. In her spare time, she enjoys rollerblading, windsurfing, watching good movies, and playing with her American Eskimo dog, Jamie.

Her present research interests lie at the intersection of computer science, biology, and medicine. Her longer term goals are to explore interdisciplinary approaches to solving problems in the medical field after attaining her Doctor of Medicine degree.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Recent years have seen a surge of new computer architectures as industry and academia work to develop faster processing power. With the predominance of high-level programming over machine-level programming as well, the need for debugging tools that use source language names and notations has increased.[1] Much effort has been given to automating the phases of compiler writing in order to simplify high-level language implementation for these new architectures. Similar efforts at automation have not, unfortunately, been given to the production of debuggers.

This lack of automation in debugger production can prove expensive in terms of engineering hours, and thus monetary costs, required for development. Early on in the development of an experimental computer system, a low-level debugger is needed to evaluate whether the system is working correctly. After the new computer system is running, each new high-level language written for the system requires a corresponding high-level debugger because users want to debug in terms of the symbols and constructs of the source language. One method for meeting these debugging needs would be to develop from scratch a new debugger for each new architecture and for each new language implemented for a given architecture. Unfortunately, writing debuggers is not only tedious but also time consuming.

[1]The terms "high-level debugging," "source-level debugging," and "symbolic debugging" are used interchangeably to mean debugging of programs in terms of their source-level names and constructs.

A similar problem confronted compiler developers about fifteen years ago. Compilation technology has since then focused on reducing duplication of effort for various phases of compiler implementation with considerable success. Most notably, parser generators[Joh75, MKR79, ASU86, FJ88], such as yacc[Joh75], and scanner generators, such as lex[ASU86, FJ88], have essentially eliminated the manual creation of parsers and scanners, respectively. Less known but also important have been efforts at automating the development of code generators[GG78, DNF79, Bir82, LJG82] and even entire compilers[BBK$^+$82, Ras82, Tof90, Sto77, Sch88]. Maygen explores the possibility of applying similar ideas of automation to debugger development.

1.1 Project Overview

This thesis explores a novel approach to providing source-level debugging support through the development of a "debugger generation system." In general, an all-purpose debugger generation system might be a tool that takes as input a source language description and a machine architecture description,[2] and produces as output a fully functional, stand-alone, language-dependent debugger for the specified architecture. Figure 1-1 depicts such a system.

A debugger produced by such a generation system consists of a core debugger skeleton (SKEL) provided by the generator, a source language interface (SLI) created by the generator from the source language input, and a machine architecture interface (MAI) created by the generator from the machine architecture input. Figure 1-2 depicts the components of such a generated debugger.

The debugger generation system designed in this project is called Maygen.[3] Maygen differs from the described all-purpose generation system in terms of what information is conveyed from each of the source language and machine architecture developers to the

[2]Details about the terms "source language" and "machine architecture" can be found in Section 4.2.

[3]The name "Maygen" originated from an initial project goal of generating various symbolic debuggers for one specific target architecture, the Mayfly[Dav92]. The project later evolved to encompass various target architectures as well, though the name Maygen remained.

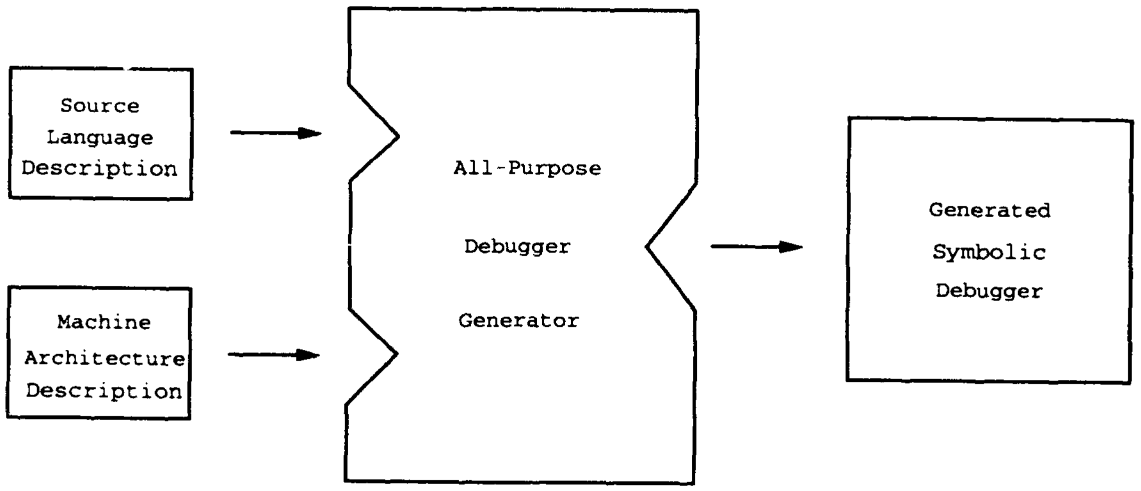
Source Language Description
Machine Architecture Description
All-Purpose Debugger Generator
Generated Symbolic Debugger

Figure 1-1: An All-purpose Debugger Generation System

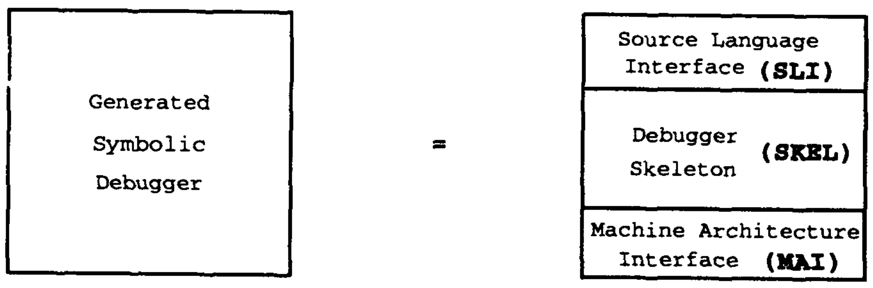
Generated Symbolic Debugger
=
Source Language Interface (SLI)
Debugger Skeleton (SKEL)
Machine Architecture Interface (MAI)

Figure 1-2: The Components of a Generated Debugger

generation system. In the all-purpose system, input consists of source language and target architecture *descriptions* that are then used by the generator to automatically create the needed interface routines. In the Maygen system, the maximal set of routines comprising each interface is fully specified by Maygen to the users of the generation system; the input from the users contains information that conveys to Maygen which of the defined interface routines are available. Once the available interface routines are known, the Maygen system determines what additional components (parts of the SKEL) are necessary to provide overall debugger functionality as well as to promote the smooth interaction of the two interfaces described above. The Maygen system framework maintains the debugger skeleton, interprets the inputs, and performs the necessary information processing to create a stand-alone, language-dependent and architecture-dependent debugger.

Figure 1-3 depicts the interrelationship among users of the Maygen system. Maygen users can be classified into one of two groups. "Phase I" users work with the Maygen system at debugger generation time, while "Phase II" users work with generated debuggers at debugger runtime.

A prototype of the Maygen system has been developed and two test sets have been run to demonstrate the viability of such a system. The test sets include a declarative Prolog-like source language running on a target virtual machine emulator and an imperative source language running on a target parallel, message-passing distributed-memory architecture.

1.2 Thesis Organization

The remainder of this thesis describes the advantages and disadvantages of related work, explains why the Maygen generated debugger is a more feasible approach, and presents the design, implementation, evaluation, and achievements of the Maygen system.

Chapter 2 begins by briefly examining previous research efforts at providing debugging support for multiple languages.

Chapter 3 presents the features of the canonical Maygen debugger in comparison and in contrast to existing debuggers.

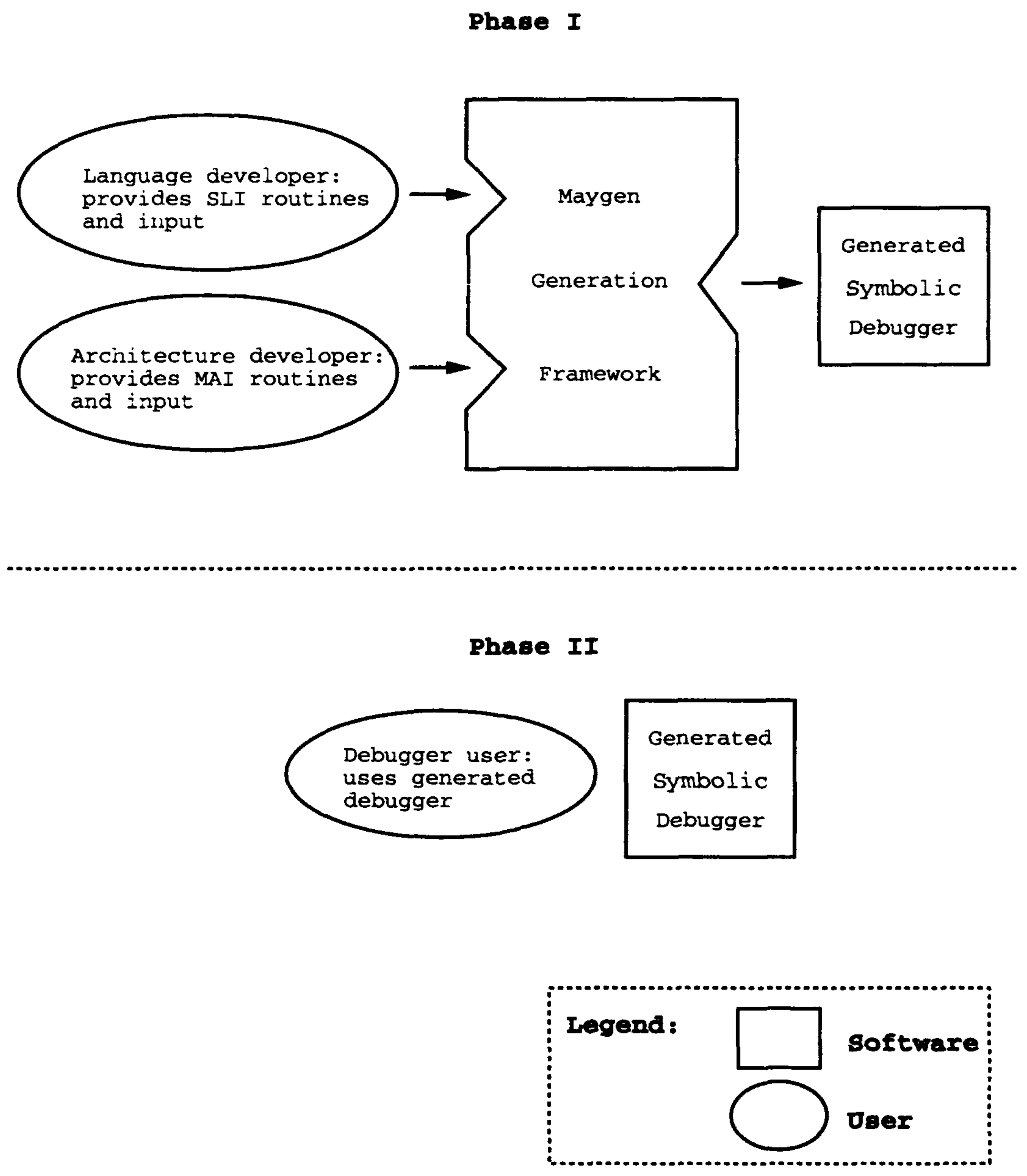
Phase I
Language developer: provides SLI routines and input
Architecture developer: provides MAI routines and input
Maygen
Generation
Framework
Generated Symbolic Debugger
Phase II
Debugger user: uses generated debugger
Generated Symbolic Debugger
Legend:
Software
User

Figure 1-3: Interrelationship Among Maygen Users

Chapter 4 then describes the Maygen system design, including the source language and machine architecture interface protocols, the core debugger skeleton, and the generation framework used to create debuggers.

Chapter 5 elaborates upon the prototype of the Maygen system that was developed, as well as provides some of the more interesting implementation issues involved.

Chapter 6 then discusses the test cases used to evaluate both the capabilities and the effectiveness of the generation system prototype.

Finally, Chapter 7 summarizes the Maygen project, presents the author's conclusions, and speculates upon possible directions for further research in the area of debugger generation.

Chapter 2

Related Work

The idea of debugger generation, although no such system is known to exist or to ever have been designed, was proposed by Johnson[Joh78] in 1978. While Johnson's own focus was on providing a multilingual tool for debugging, he commented that a debugger generation system could possibly be an alternative approach to providing source-level debugging support for multiple languages.

Despite the lack of previous work on debugger generation, two related areas of research have provided some insight for the Maygen project. Specifically, the areas of multilingual debugging and language-independent debugging also try to provide debugging support for multiple languages.

2.1 Multilingual Debugging

Multilingual debugging is a debugging style that permits the debugging of software in which components have been written in more than one source language[Joh82]. Multilingual debugging is useful to consider because of some issues that are similar to those of debugger generation. Specifically, the need to distinguish between language-dependent and language-independent components of debuggers pertains to both.

Two examples of multilingual debuggers are VAX DEBUG[Bea83] and SWAT[Car83].

VAX DEBUG is the VAX-11 Debugger developed at Digital Equipment Corporation. For a particular set of supported source languages, VAX DEBUG understands: how symbol names are composed in the language, how language expressions are interpreted, how and when type conversions are done in the language, how values in the language are displayed, and how the language scope rules work. Although VAX DEBUG understands this information for a defined set of languages, it operates according to the rules of only one language at a time. VAX DEBUG supports the following languages: assembly, Fortran, Bliss, Basic, Cobol, Pascal, and PL/I.

SWAT is a source-level debugger developed by Data General Corporation. SWAT supports five high-level languages, each of which conforms to an agreed upon "Common Compiler Component Methodology." This methodology defines a common intermediate language, procedure-calling sequence, and language runtime environment that must be followed by each of the supported languages. The languages understood by SWAT are: C, Cobol, Fortran 77, Pascal, and PL/I.

2.2 Language-independent Debugging

Similar to the idea of multilingual debugging is language-independent debugging. Language-independent debugging refers to debugging techniques that are independent of any one particular source language[Joh82]. A debugging system that has dealt specifically with the issue of language-independence is the RAIDE system[Joh77]. Johnson explains that a separate *debugging language* might be desirable. The debugging language created for the RAIDE system, called Dispel[Joh81], is designed to aid communication between an interactive user and a runtime, symbolic debugging system.

2.3 Advantages and Disadvantages

Indeed, these previous systems present approaches to debugging that appear to accommodate multiple languages. Such accommodation leads to improved economy of implementation as well as increased ease in product maintenance. In addition, these systems offer a certain amount of functional consistency to the debugger user.

Unfortunately, these systems have several shortcomings. First, they are unable to handle the peculiarities of any specific language; there is no extension mechanism with which to cater to the needs of a given particular language. Second, the languages supported by each of the multilingual debuggers are specified beforehand; to handle another language would mean having to rewrite the debugger itself. These systems are limited to debugging not just a pre-defined set of languages, but moreover, only a pre-defined set of semantically similar languages.

A further fault lies in the language-independent debugging system as well. A user must first learn a completely separate language, the debugging language, before even being able to start debugging a program. Once debugging can actually proceed, the user then needs to worry about the possibility of faulty *debugging programs* in addition to faulty source programs.

Admittedly, multilingual and language-independent debugging techniques offer some gains over single-language debuggers. Nevertheless, the deficiencies in these debugging techniques are considerable.

Chapter 3

Canonical Generated Debugger

The Maygen debugger tries to maintain the desirable features of multilingual and language-independent debuggers while also trying to improve upon their shortcomings. This chapter begins by describing the features of the canonical Maygen generated debugger, proceeds to explain the motivation behind the chosen design, and then demonstrates how this design is able to offer more than multilingual and language-independent debuggers.

3.1 Overview

The canonical Maygen debugger generally resembles a typical single-language source-level debugger for a compiled language in that it offers the "traditional" functionality with which users are accustomed to debugging programs. The Maygen debugger debugs compiled code that has not been optimized. It is also expected that the user starts up the Maygen debugger and then runs a program under debugger control. The maximal set of fundamental debugging facilities that are supported[1] by a Maygen debugger include: starting, stopping, single-stepping, and continuing an execution; loading a file; resetting the machine; setting, clearing, and listing machine-level as well as source-level breakpoints; activating and sus-

[1]Each of the supported facilities is only available upon satisfaction of specific conditions. See Chapter 4 for details.

Table 3.1: CANONICAL MAYGEN DEBUGGER FUNCTIONALITY

Start execution
Stop execution
Continue execution
Single-step execution (following calls)
Single-step execution (not following calls)
Load a file
Reset the machine
Set, clear, list machine-level breakpoints
Set, clear, list source-level breakpoints
Activate breakpoints
Suspend breakpoints
Display and set variable values
Display register values
Trace and untrace variables
Trace and untrace procedures
List traced variables
List traced procedures
List user program labels and symbols
Show current source line
Print information about debugger status
Display list of debugger commands
Repeat previous command
Quit Debugger
Comment (ignored)

pending breakpoints; displaying and setting variable values and register values; tracing and untracing variables and procedures; listing traced variables and procedures; indicating the current source line; displaying a list of debugger commands with help information; repeating the previous command; quitting the debugging session; and adding a comment. The Maygen debugger functionality is summarized in Table 3.1.

Each command's availability depends upon its semantic correctness in the context of the particular source language or machine architecture involved, as well as upon the support provided by both the source language and the machine architecture developers. For example, a debugger user should not be able to set logic variables in Prolog; thus, the command to set the value of a variable is not made available in a generated Prolog debugger. In this manner, each generated debugger is tailored specifically to the particular language and architecture in question.

In addition to the fundamental debugging facilities, the Maygen debugger also has a mechanism for incorporating extension commands that are then fully available to the debugger user. For example, the option to choose whether an execution will proceed in a breadth-first manner or a depth-first manner is not provided by the canonical Maygen debugger; however, this might be a desirable command to have in a Prolog debugger. A Prolog system developer, then, can specify this option as an extension command to the Maygen system, which will then add it to the set of commands available in the generated Prolog debugger.

Extension commands can be specified and provided by the source language developer, the machine architecture developer, or both. Extension commands are of two general flavors. "Independent" extension commands are self-contained in that their functionality does not depend upon any routines that might not be available, e.g., from either the source language interface routine set or the machine architecture interface routine set. "Dependent" extension commands, on the other hand, are not self-contained in that their functionality, and thus their availability to the debugger user, depends upon at least one of the routines from either the source language interface routine set or the machine architecture interface routine set.[2]

Finally, the canonical Maygen debugger understands that not all machines are uniprocessors; the Maygen debugger understands that a machine may have more than one processing node. In such cases, the Maygen debugger operates on a single node at a time. The debugger user has the ability to determine the total number of processing nodes present, determine

[2]Either type of extension command—independent or dependent—can use routines explicitly provided by the debugger skeleton if desired. (See Chapter 4 for details.) Since the availability of an extension command does not hinge upon the availability of routines provided by the debugger skeleton (because the latter are always available), debugger skeleton routines do not play a role in the classification of extension commands into one of the two categories.

Table 3.2: ADDITIONAL MAYGEN DEBUGGER FUNCTIONALITY FOR MULTIPROCESSORS

Display number of nodes present and available
Show current node
Switch to a different node
Change number of nodes available

the number of nodes available, determine which node is being debugged, switch from the current node being debugged to a different node, and change the number of nodes available. Maygen's default mode of execution for multiprocessors is that which is provided by the machine architecture developer. Table 3.2 summarizes the additional debugger functionality provided by Maygen for multiprocessor architectures.

3.2 Design

3.2.1 Debugging Unoptimized Compiled Code

The canonical Maygen debugger was developed to work on unoptimized, compiled code rather than on optimized or interpreted code. Although using an interpreter as the base of a debugger might be beneficial because of how well it supports interactive debugging[Mak91], the approach is more complicated. In addition to a debugger skeleton, the generation system would need to maintain an interpreter skeleton as well. This interpreter skeleton either would need to interpret a broad class of source languages, which is currently infeasible[Joh77], or would need to be developed by the generation system into a language-dependent, architecture-dependent interpreter. The generation of such an interpreter might itself be an interesting research problem, but is tangential to the issue of debugger generation.

Furthermore, Troisi[Tro82] points out that interpreted code may run differently than compiled code; thus, a debugger based upon an interpreter may not illuminate the problem area of the source code. In addition, a debugger based upon an interpreter might suffer from significantly decreased execution speed[Edw75].

Likewise, the issue of debugging optimized code is also tangential to the primary concern of how to automatically create a symbolic debugger.[3] Thus, the canonical Maygen debugger expects that the code a user loads and therefore wants to debug is unoptimized. Once such code has been determined to be correct, then the user can explore performance issues.

3.2.2 Providing Tailored, Traditional Functionality

The canonical Maygen debugger offers a variety of traditional debugging commands to the user. Such a design was chosen not only because users are more accustomed to this method of debugging and thus can have less startup time learning how to use a Maygen debugger, but also because users would be provided with the essentials of a runtime debugging system, which are the ability to set breakpoints and examine values within the program being debugged[Bro79, Joh81].

Some traditional debugging commands, such as starting an execution, make sense for essentially all languages. The relevance of some other commands, however, are not necessarily immediately apparent. For example, setting a breakpoint makes perfect sense in a language such as C or Pascal; but, what does it mean to set a breakpoint in Prolog? It might, for example, mean the ability to temporarily stop execution at any of the four ports of the multiported box model for Prolog execution[SW90]. Another example is the tracing of variables. This might make good sense in an imperative language, but what does it mean in a declarative one? An example of how the tracing of variables could be used in a declarative language is to follow clauses that match (are true) for a particular search. In cases such as the two described, it is left up to the language developer or architecture developer to decide in what manner each supported traditional debugging command can be best exploited for debugging of the given language on the given architecture.

[3]See Section 7.2.2 for more details.

3.2.3 Supporting Extension Commands

Admittedly, not all of the traditional debugging commands are necessarily applicable for all source languages or all machine architectures. For this reason, the Maygen debugger might only provide a subset of the traditional commands, depending on the specific language and architecture in question. That is, the Maygen debugger is specifically designed to be capable of having a command set tailored to the target language and architecture.

This tailoring of the Maygen debugger's command set goes beyond simply deleting irrelevant or inapplicable traditional debugging commands. Such a system would be not only too limiting for the extremely unconventional target language and/or architecture, but also not good enough for a more conventional but slightly different target language and/or architecture. Accordingly, the Maygen debugger is designed to support extension commands. The extension commands enable language and architecture developers to extend the basic command set of a Maygen debugger to include any additionally desired functionality that is potentially highly-specific for that particular language or architecture.

3.2.4 Supporting Multiprocessors

Although the target architecture for Maygen might be a parallel one, the focus of this project is on developing a method for *generating debuggers* rather than on determining the best way to *implement a parallel debugger.* Thus, Maygen debuggers have been designed to deal only with simple notions of parallelism, such as knowing about the existence of multiple processing nodes. A Maygen debugger operates on one processing node at a time and can switch from one node to another upon the user's request. These capabilities allow for more meaningful debugging on a multiprocessor than possible from a debugger with absolutely no knowledge of multiple nodes. Maygen generated debuggers do not, however, address more complex parallelism issues, such as the monitoring of interprocess communication. Such issues, although potentially beneficial, would tend to detract from the primary concern of the project.

3.3 Advantages

The more obvious advantages of using Maygen debuggers over traditional, single-language debuggers are similar to the advantages attributed to the use of multilingual or language-independent debugging techniques. First, Maygen debuggers still present a certain degree of functional consistency to the debugger user, resulting in less learning overhead. Second, Maygen debuggers are cheap to build since they require little work on the part of language developers and architecture developers compared to the effort needed to create debuggers from scratch. Finally, maintenance is simplified because the driving engine of the debugger is similar from one Maygen debugger to the next.

While Maygen debuggers share the advantages of multilingual and language-independent debugging systems over traditional, single-language debuggers, Maygen debuggers additionally compensate for the deficiencies inherent in multilingual and language-independent systems. Maygen debuggers are flexible; they can be tailored to the specific needs and peculiarities of different languages and architectures. This flexibility comes in part from the selective availability of the supported debugging routines. More importantly, though, this flexibility comes from the system's allowance of and support for extension commands. These features taken together result in a system capable of handling semantically different languages. Furthermore, Maygen debuggers can be generated for more than just a pre-defined, limited set of languages.

How is it that the Maygen debugger can be so flexible? The answer lies in the fact that it is a *generated* debugger, that it is generated according to the specifics of each particular language and each particular architecture. This is made possible through the Maygen generation system.

Chapter 4

Generation System Design

4.1 Overview

The Maygen system consists of three major components: a set of interface protocols, a debugger skeleton, and a generation framework. The protocols specify the exact nature of the interface routines that promote smooth communication between the debugger skeleton and the rest of the programming environment.[1] The routines that are available for a given debugger to be generated are conveyed by way of input files to the generation framework. The generation framework, housing the debugger skeleton, processes the input data and produces a stand-alone, language-dependent and architecture-dependent debugger.

Figure 4-1 portrays the components of the Maygen system and how they are interrelated, while Figure 4-2 shows the pieces of a Maygen generated debugger.

The Maygen system was designed in this manner in order to have the capability of producing a debugger that is flexible, in terms of handling very different inputs, yet practical, in terms of providing large savings to language and architecture developers. Since interpreter-based debuggers have some intrinsic problems, the debugging of compiled code was chosen as the basis for Maygen. The decision to have a generation system at all evolved

[1] The "rest of the programming environment" refers to the "source language" and "machine architecture." These are explained in detail in Section 4.2.

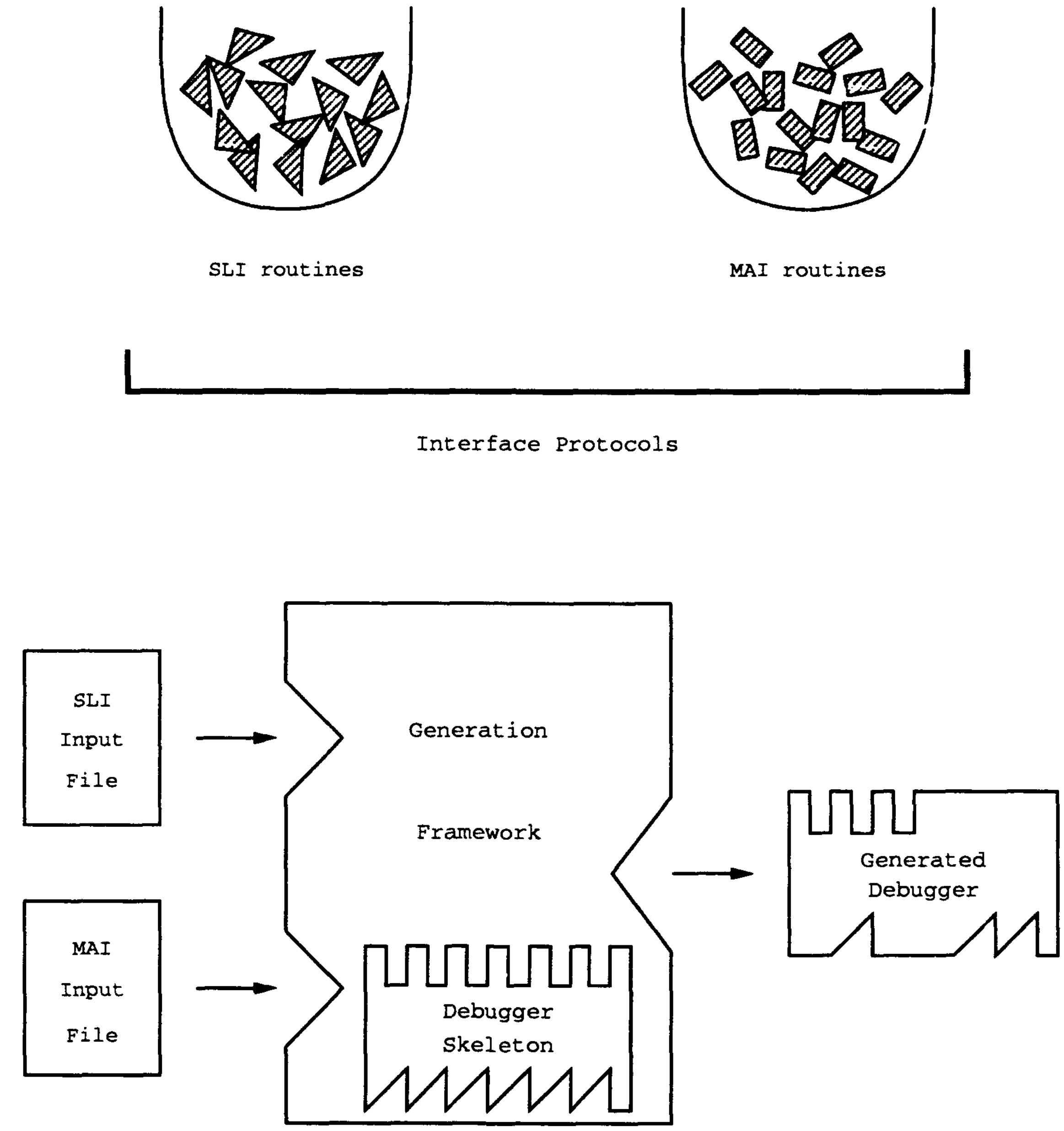

Figure 4-1: The Maygen Debugger Generation System

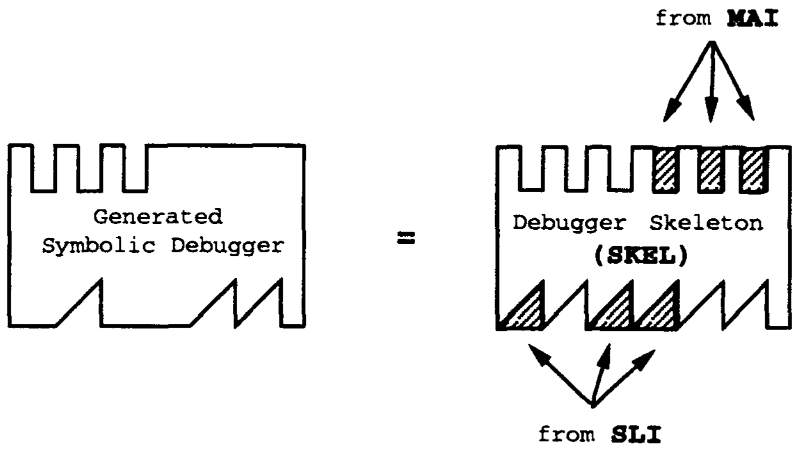

Figure 4-2: THE COMPONENTS OF A MAYGEN GENERATED DEBUGGER

from the knowledge that non-generated debuggers, such as multilingual debuggers, lack the flexibility needed to support an arbitrary number of language systems as well as to handle semantically different language systems. On the one hand, the generation aspect, tailoring ability, and extension mechanism of the Maygen system make canonical Maygen debuggers flexible. On the other hand, the core debugger skeleton along with the automatic processing of it into a generated debugger make canonical Maygen debuggers practical.

An alternative method that was considered for achieving the dual goals of flexibility and practicality was to add debugging constructs to a source file in a preprocessing-type step. Preprocessors have the advantage that the compiler of the source language to be debugged need not be modified[Edw75]. This method, however, seemed to be extremely limiting in terms of what debugging capabilities a debugger user would have, as well as in terms of what languages and systems could actually be handled effectively.

4.2 Interface Protocols

An important aspect of developing the Maygen system is deciding upon the interaction of the Maygen debugger with the rest of the world. Some programming languages employ the notion of an abstract machine, or virtual machine, with which to serve conceptual[2] and/or implementational[3] purposes. When this is the case, the high level aspects of the abstraction could be exploited for the purposes of debugging. An example is the modification of the ports of the Prolog box model to support debugging[SW90].

Conventional languages such as C and Fortran do not really have abstract machines with which to visualize their execution. For example, in a Unix system[MM83], an object file produced by the C compiler executes as just another process running under the Unix operating system. Conceptually, one might visualize that process having a certain amount of memory allocated to it and have a notion of data and instructions residing in that memory, as well as a "location counter" that indicates the current instruction being executed. Clearly, such a mental model of program execution is down near the level of the operating system and machine architecture on which the process is running.

The Maygen system adopts an intermediate position toward debuggers that attempts to take advantage of higher levels of abstraction when available, but that can be used for lower-level conventional programs as well. The Maygen system separates the *source program* from the *evaluation environment*.

Accordingly, the two interfaces to the Maygen debugger are the source program and the evaluation environment. The interface to the source language is fittingly referred to as the Source Language Interface (SLI). The interface to the evaluation environment is less appropriately referred to as the Machine Architecture Interface (MAI); this interface might

[2]As a conceptual technique, the abstract machine allows a high level way to think about the execution of a program. This capability is especially useful when the programming language contains non-trivial control mechanisms such as Prolog's unification or Snobol's pattern matcher.

[3]As an implementation technique, the abstract machine can serve as a specification that describes details of a particular algorithm, such as a unifier or pattern matcher, used to implement the language. In addition, the abstract machine can serve as an implementation prototype, as in the Lisp functions Eval and Apply, which define the complete Lisp evaluator in just a few lines of Lisp code.

encompass not only the machine architecture, but also a runtime system, an operating system, an abstract machine, or a combination.

The interface protocols specify the exact nature of the routines that are used by the core debugger to interact with the source program and the architecture.[4] Each interface protocol can be thought of as the set of routines that comprise the interaction between the core debugger and source program, or between the core debugger and machine architecture. The Source Language Interface routines are provided by a language developer, while the Machine Architecture Interface routines are provided by a system developer.

Each interface consists of approximately fifteen routines; these translate to the supported functionality of a generated debugger. There exists a minimal subset of routines that are required of the Source Language Interface and of the Machine Architecture Interface in order for a working debugger to be generated. With the provision of this minimal subset, Maygen can automatically create a low-level debugger. With the provision of increasingly more Source Language Interface and Machine Architecture Interface routines, Maygen can create symbolic debuggers with increasingly larger amounts of functionality. These sets of interface routines are experimentally derived.

Table 4.1 lists the routines constituting the Source Language Interface as specified by the current Maygen design. Similarly, Table 4.2 lists the routines contained in the Machine Architecture Interface as specified by the current Maygen design.

The interface protocols not only specify the routines that should be provided, but also the format in which such information is conveyed to the generation framework. The input to the generation framework consists of two text files, one for information about the Source Language Interface and the other for information about the Machine Architecture Interface. The Source Language Interface input file contains: a listing of the Source Language Interface routines with specification of whether or not each is available, the name of the source language, the location and name of a library containing the Source Language Interface

[4]Henceforth, the "machine architecture" and the "architecture" refer to the evaluation environment, except when specified otherwise.

Table 4.1: Source Language Interface Routines

Initialize SLI
Map procedure to object line
Map procedure beginning to object line
Map procedure ending to object line
Trace procedure
Map source line to object line
Read in symbols
Print labels
List procedures
Print symbols
Display text of current source line
Untrace procedure
Process initial debugger arguments
Print SLI information

Table 4.2: Machine Architecture Interface Routines

Initialize MAI
Is program loaded?
Install machine breakpoint
Continue program
Uninstall machine breakpoint
Set machine breakpoint on a procedure
Clear machine breakpoint on a procedure
Read in program
Print register contents
Run program
Step, following procedure calls
Step, not following procedure calls
Reset machine
Process initial debugger arguments
Print MAI information
Change current processing node
Change number of available nodes

routines, and information about each extension command desired by the language developer. This extension command information includes the total number of extension commands supported by the language developer as well as details about each extension command. These details include: the name of the command, the declaration used to indicate it is an externally defined procedure, the invocation of the command with its arguments, and a list of Source Language Interface and Machine Architecture Interface routines upon which the proper functioning of the extension command depends.[5]

Similarly, the Machine Architecture Interface input file contains: a listing of the Machine Architecture Interface routines with specification of whether or not each is available, the name of the architecture or abstract machine, the location and name of a library containing the Machine Architecture Interface routines, and information about each extension command desired by the machine developer. The information for these extension commands is exactly analogous to that of the extension commands for the Source Language Interface. The Machine Architecture Interface input file additionally contains information about how many processing nodes are present as well as how many processing nodes are available in the target architecture.

An example of a Source Language Interface input file template, which can be filled in by a language developer, can be found in Appendix A. Appendix B contains an example of a Machine Architecture Interface input file template.

4.3 Debugger Skeleton

The debugger skeleton consists of the components of a symbolic debugger that have been determined to be language-independent and architecture-independent. These components have been grouped together to form the *core* of a debugger, hence debugger *skeleton*, which the Maygen system uses as the backbone with which to create Maygen debuggers.

The debugger skeleton can be thought of as providing the glue necessary for coherently

[5]For independent extension commands, this list will be empty.

sticking together the interface routines. More accurately, the debugger skeleton is several files of code, some of which contribute directly (unchanged) to the code of a generated debugger, and some of which are either supersets of or incomplete fragments of code that will be modified by the generation framework into code that will then be part of a generated debugger. The final output files include a makefile with which the user can make the newly-generated debugger from its source code.

More descriptively, the debugger skeleton consists of debugger components such as the debugger user interface, command loop driver, and grungy initialization and maintenance routines, e.g., for keeping track of tracing. The debugger user interface can range from a simple textual interface to a much more elaborate graphical user interface. This interface need only be written once and then can be used for each subsequent Maygen debugger. An example of a grungy maintenance job is the breakpointing facility: coordinating the setting (and checking for duplicates), clearing (and checking for validity), keeping track, listing, installing, uninstalling, activating, and suspending of machine-level and source-level breakpoints.

Each debugger command supported by the debugger skeleton is affiliated with certain Source Language Interface and Machine Architecture Interface routines upon which its functionality depends. A given, supported debugger command is only available if the routines upon which it depends are made available by the language and/or architecture developers. For example, the command that allows a debugger user to set a breakpoint on a source line depends upon one Machine Architecture Interface routine (install machine breakpoint) and one Source Language Interface routine (map source line to object line). If either of these routines is not supported, then the source-level breakpoint setting command is unavailable in the subsequently generated debugger. The debugger commands supported by the debugger skeleton are identical to those previously described in Table 3.1.

As mentioned previously, a few debugger skeleton routines are explicitly provided to aid Maygen system users. Language or architecture developers can freely call these routines from within either extension commands or Interface routines. The debugger skeleton

Table 4.3: Debugger Skeleton Routines Available to Developers

Install breakpoints
Uninstall breakpoints
Check whether breakpoint address already exists
Add procedure to list of procedure breakpoints
Remove procedure from list of procedure breakpoints
Add machine address to list of machine breakpoints
Remove machine address from list of machine breakpoints

routines supported in this manner are listed in Table 4.3.

4.4 Generation Framework

This section describes the overall framework used by the Maygen system to create a functional debugger. This framework serves as the driving engine for accepting input information about the Source Language and Machine Architecture Interfaces, for translating the input into which debugger commands will be available, and for appropriately modifying and appending the debugger skeleton to make it a stand-alone debugger.

The generation framework understands the format of the input files and thus can read and interpret the information in the input. The generation framework also houses, or more accurately, keeps track of, all the pieces of the debugger skeleton. The framework knows which pieces are to be left intact to become part of a generated debugger as well as which need to be either augmented or chopped and spliced.

The generation framework decides, based upon which Source Language Interface and Machine Architecture Interface routines are known to be available, what components will go into the debugger to be generated and how these components should be put together to make a working unit. The framework processes the input information to determine which debugger commands will comprise the command set of the debugger to be generated. These command names are then incorporated into the "help list" available to debugger users, while the code that implements these commands are incorporated into the source code files that compile into the functional debugger. Finally, the generation framework outputs all the necessary code files and a makefile for the new Maygen debugger.

The framework is designed to perform *at generation time* all of the interpretation and processing necessary for a given debugger to be generated. By performing all input interpreting and processing during debugger generation, Maygen debuggers can avoid unnecessary runtime inefficiency.

Chapter 5

Prototype Implementation

The Maygen system design encompasses more than does the prototype that has been implemented thus far. This chapter describes the environment in which the system was developed and the scope of the prototype, as well as presents some of the more interesting implementational details.

5.1 Overview

The experiment was carried out using the equipment and facilities of Hewlett Packard Laboratories. A single-processor workstation HP9000/840 running HP-UX 7.0, Hewlett Packard's version of UNIX, was used for the development of the debugger generation system. The prototype Maygen system is written in the C language.

The prototype generation system consists of the Source Language Interface and Machine Architecture Interface protocols with routines defined and input file formats specified, an implemented subset of the designed debugger skeleton, and a functional generation framework that handles the existing debugger skeleton and inputs.

5.2 Maygen Debugger Features

The canonical Maygen debugger of the prototype generation system supports most of the functionality supported by that of the designed system. These commands are summarized in Table 5.1. The commands that are not supported in this implementation are listed in Table 5.2. An additional note is that the support for tracing and untracing of procedures is currently implemented as the setting and clearing of breakpoints on procedure names. Tracing of procedures could be made more elaborate by not only breaking when a procedure is reached, but also automatically displaying the values of the procedure's arguments upon invocation and displaying any return value upon exit.

As in the design, each debugging command's availability depends upon its semantic correctness in the context of the particular source language or machine architecture involved, as well as upon the support provided by both the source language and the machine architecture developers.

The prototype canonical Maygen debugger is able to support one of the two flavors of extension commands described in Section 3.1. Independent extension commands are currently incorporated in the prototype, whereas dependent extension commands are not.

Finally, the prototype Maygen debugger operates on a single node at a time, but understands that there might be more than one processor in the target architecture. Thus, when the target architecture has multiple nodes, the generated Maygen debugger allows the user to: determine the total number of nodes present, determine how many nodes are available, find out which node is being debugged, switch between nodes, and change the number of nodes available. This functionality is identical to that designed, which is summarized in Table 3.2.

5.3 Interface Protocols

The Source Language Interface and Machine Architecture Interface are implemented as described in Section 4.2, having the goal of separating the source program from the evalua-

Table 5.1: DEBUGGER FUNCTIONALITY IMPLEMENTED IN PROTOTYPE

Start execution
Stop execution
Continue execution
Single-step execution (following calls)
Single-step execution (not following calls)
Load a file
Reset the machine
Set, clear, list machine-level breakpoints
Set, clear, list source-level breakpoints
Activate breakpoints
Suspend breakpoints
Display register values
Trace and untrace procedures
List user program labels and symbols
Show current source line
Print information about debugger status
Display list of debugger commands
Repeat previous command
Quit Debugger
Comment (ignored)

Table 5.2: DEBUGGER FUNCTIONALITY NOT IMPLEMENTED IN PROTOTYPE

Display and set variable values
Trace and untrace variables
List traced variables
List traced procedures

tion environment. The specified routines comprising the Source Language Interface are the same as those listed in Table 4.1; likewise, the specified routines comprising the Machine Architecture Interface are the same as those enumerated in Table 4.2.

The input file formats, which the Maygen prototype uses, are identical to those prescribed by the interface protocol design of Section 4.2. The sample Source Language Interface input file template located in Appendix A is the actual input file template used for the prototype's language test cases. Similarly, the sample Machine Architecture Interface input file template located in Appendix B is the actual input file template used for the prototype's architecture test cases.

5.4 Debugger Skeleton

The prototype debugger skeleton consists of components of a symbolic debugger that are language-independent and architecture-independent, as designed. However, the prototype debugger skeleton does not encompass as much basic supported functionality as does the designed debugger skeleton. Also, the debugger user interface is a purely textual one.

The command loop driver is based upon a C language `switch` statement that switches on the interactive user's typed command. This implementation was chosen for relative efficiency in carrying out the desired command and for ease in tailoring the appropriate code files to the inputs.

The debugger skeleton consists of five files that contribute unchanged to a generated debugger's source code and six files that are modified into files that are then directly part of a generated debugger's source code. The files that contribute unchanged contain source code files that implement breakpoints, essential debugger initializations and driver routines, and input/output routines. These files also include header files that list Source Language Interface, Machine Architecture Interface, and debugger skeleton routines.

The files that need to be modified before becoming part of a generated debugger are the makefile, "cases" file, "filler" file, extension command file, "miscellaneous" file, and "debugger help list" file. The "cases" file is a superset of the code needed to decide what to perform for each command. When the prerequisite routines are available for a given debugger command, that command will be associated with code that performs the actual command; when the prerequisite routines are not available, however, that command will be associated with code that relays to the user the unavailability of the invoked command. In addition, each command is accordingly added or not added to the debugger help list in the "debugger help list" file. Thus, when a user calls up a help list of debugger commands, those commands that are not available, due to lack of sufficient support from either the language or architecture developer, will not be included in the list. The "filler" file is created by Maygen to account for all of the Source Language Interface and Machine Architecture Interface routines that are not provided as inputs. Maygen creates "filler" routines to satisfy the compiler's checks, knowing that these dummy routines will not actually be called. The extension command file is created by Maygen to handle the calling of appropriate extension commands upon a user's invocation of such commands. Finally, the "miscellaneous" file is created by Maygen to hold two architecture-dependent definitions as well as routines for printing information upon debugger startup and exit.

5.5 Generation Framework

The prototype generation framework is as described in Section 4.4. This generation framework understands the input file formats, reads and interprets the input files, accordingly performs the actual modifying of the debugger skeleton files described in the previous section, and outputs all necessary source code to create a new debugger.

Chapter 6

Results

This chapter discusses the test cases used to evaluate the prototype generation system, and hence the Maygen system design itself.

6.1 Overview

The goal for choosing the test cases was to select domains that are quite different in order to show the flexibility that Maygen has in comparison to existing systems for providing debugging support to multiple programming environments. Each test set[1] is comprised of a source language that conforms to the Source Language Interface protocol (in terms of interface routines and Maygen input file), and a machine architecture that conforms to the Machine Architecture Interface protocol (in terms of interface routines and Maygen input file).

Two such test sets have been run through the Maygen system. The two source languages and their evaluation environments are: a declarative language, OPAL, running on the OM virtual machine, and an imperative language, C, running on the Mayfly parallel architecture. By generating a symbolic debugger for both a declarative language and an imperative language, Maygen demonstrates its ability to handle semantically-different languages.

[1]A "test set" consists of both a source language "test case" and a machine architecture "test case."

Table 6.1: SLI ROUTINES SUPPORTED BY OPAL

Initialize SLI
Map procedure to object line
Map procedure beginning to object line
Read in symbols
Print labels
Print symbols
Print SLI information
Process initial debugger arguments

6.2 Test Cases

6.2.1 OPAL and OM

OPAL, the Oregon Parallel Logic language, is a Prolog-like language developed at the University of Oregon[Con90, Con91, Con92]. OPAL is based on the AND/OR Process Model[Kac90], which is an abstract model for parallel logic programs. The AND/OR Process Model has an operational semantics defined by asynchronous objects that communicate entirely by messages.

OPAL programs are compiled into the instruction set of the OPAL Machine, or *OM*. The OM is a virtual machine similar to the Warren abstract machine[War83] for standard Prolog implementations. The difference is that the OM virtual machine is designed for programs that execute according to the AND/OR Process Model on nonshared memory multiprocessors. The version of the OM virtual machine used for this test set runs on a uniprocessor UNIX workstation; it does not exploit AND or OR parallelism in this implementation.

The OPAL language test case supports eight out of the fourteen Source Language Interface routines specified by the Maygen prototype and provides no extension commands. The routines supported by OPAL are summarized in Table 6.1, while those that are not supported are listed in Table 6.2.

The OM virtual machine test case supports fifteen out of the seventeen Machine Archi-

Table 6.2: SLI ROUTINES NOT SUPPORTED BY OPAL

Map procedure ending to object line
Trace procedure
Map source line to object line
List procedures
Display text of current source line
Untrace procedure

tecture Interface routines specified by the Maygen prototype. Additionally, the OM test case provides twelve independent extension commands.

The OM virtual machine supports all of the Machine Architecture Interface routines except the two routines specific to multiprocessors since the OM implementation is for a uniprocessor. Tables 6.3 and 6.4 summarize those routines supported and not supported, respectively, by the OM virtual machine.

The extension commands provided by the OM virtual machine provide the debugger user with the capabilities to choose between: searching for all solutions or for just one solution, performing a breadth-first or a depth-first search, executing in quiet mode or not, tracing processes or not during execution, tracing instructions or not during execution, and displaying registers symbolically or not. The extension commands also enable the user to print sections of object code, sections of the heap being used by the OM virtual machine, message or process information, queue contents, and a process tree for the execution. These additional features are summarized in Table 6.5. A sample OM Machine Architecture Interface input file can be found in Appendix C.

The Maygen generation framework accepted the input files of the described test set and produced a symbolic debugger for OPAL running on the OM virtual machine. The debugger commands supported by the generated OPAL debugger are listed in Table 6.6

The OPAL Source Language Interface input file and the OM Machine Architecture Interface input file were tested to have varying numbers of interface routines available to

Table 6.3: MAI ROUTINES SUPPORTED BY OM

Initialize MAI
Is program loaded?
Install machine breakpoint
Continue program
Uninstall machine breakpoint
Set machine breakpoint on a procedure
Clear machine breakpoint on a procedure
Read in program
Print register contents
Run program
Step, following procedure calls
Step, not following procedure calls
Reset machine
Print MAI information
Process initial debugger arguments

Table 6.4: MAI ROUTINES NOT SUPPORTED BY OM

Change current processing node
Change number of available nodes

Table 6.5: MAI Extension Commands Provided By OM

Toggle all-solutions
Toggle breadth-first search
Toggle quiet mode
Toggle process trace
Toggle instruction trace
Toggle symbolic register display
Print code
Print heap
Print message information
Print process information
Print queue contents
Print process tree

Maygen. The supported functionality of each resulting OPAL debugger variant was checked to ascertain that the debuggers changed accordingly. These generated OPAL debugger variants were then tested on a suite of OPAL programs to verify their correctness.

6.2.2 C and Mayfly

The language of the second test set is C, the familiar, imperative language developed by Ritchie[KR88, KW91]. C is a relatively low-level, general-purpose programming language. While C provides data types and fundamental control-flow constructions such as looping and decision making for single-threaded control flow, it does not provide built-in higher-level mechanisms such as input/output facilities or operations on composite objects such as lists and arrays.

Compiled C programs are processed by the Mayfly architecture[Dav92]. The Mayfly, developed at Hewlett Packard Laboratories, serves as a back-end processor for a Hewlett Packard Series 800 workstation. The Mayfly is a scalable, general-purpose parallel processing architecture; it is a distributed memory machine with communication supported by message passing.

Table 6.6: FUNCTIONALITY OF THE GENERATED OPAL DEBUGGER

Print help information
Repeat previous command
Activate breakpoints
Set breakpoint on object line
Set procedure breakpoint (trace procedure)
Continue from breakpoint or step
Delete breakpoint on object line
Delete procedure breakpoint (untrace procedure)
Read in compiled user program
Display general registers
Print information about debugger status
List breakpoints
List user program labels
List user program symbols
Quit debugger
Run program
Single step (follow calls)
Single step (do not follow calls)
Suspend breakpoints
Reset machine to startup state
Comment (ignored)
Execute an extension command:
- Toggle all-solutions
- Toggle breadth-first search
- Toggle quiet mode
- Toggle process trace
- Toggle instruction trace
- Toggle symbolic register display
- Print code
- Print heap
- Print message information
- Print process information
- Print queue contents
- Print process tree

Table 6.7: SLI Routines Supported By C

Initialize SLI
Map source line to object line
Map procedure to object line
Map procedure beginning to object line
Map procedure ending to object line
List procedures
Read in symbols
Process initial debugger arguments
Print SLI information

Table 6.8: SLI Routines Not Supported By C

Trace procedure
Untrace procedure
Print labels
Print symbols
Display text of current source line

The C language test case supports nine out of the fourteen Source Language Interface routines specified by the Maygen prototype and provides no extension commands. The routines supported by C are summarized in Table 6.7, while those that are not supported are listed in Table 6.8.

The Mayfly architecture test case supports sixteen out of the seventeen Machine Architecture Interface routines specified by the Maygen prototype. The Mayfly test case supports all of the Machine Architecture Interface routines except execution stepping that does not follow procedure calls. Tables 6.9 and 6.10 summarize those routines supported and not supported, respectively, by the Mayfly test case.

Additionally, the Mayfly test case provides three independent extension commands that

Table 6.9: MAI Routines Supported By Mayfly

Initialize MAI
Is program loaded?
Install machine breakpoint
Continue program
Step, following procedure calls
Uninstall machine breakpoint
Set machine breakpoint on a procedure
Clear machine breakpoint on a procedure
Read in program
Print register contents
Run program
Reset machine
Process initial debugger arguments
Print MAI information
Change current processing node
Change number of available nodes

Table 6.10: MAI Routine Not Supported By Mayfly

Step, not following procedure calls

give users the capability to select which CPU of the current processing node to debug. Each Mayfly processing node has two CPUs: the Message Processor (MP) and the Execution Processor (EP). The Mayfly extension commands provide the debugger user with the following capabilities: to select the MP of the current node for debugging, to select the EP of the current node for debugging, and to determine which CPU is the current (being debugged) CPU of a given Mayfly processing node. These additional features are summarized in Table 6.11.

The Maygen generation framework accepted the input files of the described test set

Table 6.11: MAI Extension Commands Provided By Mayfly

Select MP of current node
Select EP of current node
Determine which CPU is current CPU

and produced a C debugger for the Mayfly. The debugger commands supported by the generated C debugger are listed in Table 6.12

The C Source Language Interface input file and the Mayfly Machine Architecture Interface input file were tested to have varying numbers of interface routines available to Maygen. The resulting C debugger variants were inspected to ensure that their set of supported functionality changed accordingly. As observed for the OPAL/OM test set, the supported functionality of each resulting generated C debugger also correctly reflected the changed Maygen inputs.

Due to logistical difficulties,[2] the generated C debugger variants were "tested" by closely watching the commands attempted to be written to the Mayfly monitor, the software that connects the Mayfly architecture with its front-end workstation. Interfacing to this monitor is the Mayfly's debugger library. Normally, any debugger for the Mayfly calls basic routines from this debugger library. The debugger library routines, which normally communicate directly with the Mayfly via the monitor program, were replaced during testing with verbose stubs. Attempted command writes to the monitor from generated C debugger variants were then compared with the attempted command writes of similar debugging commands invoked from an existing, tested debugger for the Mayfly.

[2]The Mayfly architecture can only be used locally because its software currently does not support remote access. Maygen work, however, was completed 3000 miles from the residence of the Mayfly.

Table 6.12: FUNCTIONALITY OF THE GENERATED C DEBUGGER

Print help information
Repeat previous command
Activate breakpoints
Set breakpoint on source line
Set breakpoint on object line
Set breakpoint at procedure beginning
Set breakpoint at procedure exit
Set procedure breakpoint (trace procedure)
Continue from breakpoint or step
Delete breakpoint on object line
Delete breakpoint on source line
Delete procedure breakpoint (untrace procedure)
Read in compiled user program
Display general registers
Print information about debugger status
List breakpoints
List procedures
List traced procedures
Quit debugger
Run program
Single step (follow calls)
Suspend breakpoints
Reset machine to startup state
Comment (ignored)
Execute an extension command:
- Select MP of current node
- Select EP of current node
- Determine which CPU is current CPU
Execute a multinode command:
- Change processing nodes
- Determine current number of nodes
- Determine current node

Chapter 7

Conclusions

This chapter summarizes the Maygen project, presents some conclusions about debugger generation in general and the Maygen approach in specific, and suggests areas for further research.

7.1 Summary

The ability to provide debugging support for multiple languages is an important one because of today's demand for high-level debuggers to accompany high-level languages.

Two previous approaches that were considered for providing debugging support for multiple languages are multilingual debugging and language-independent debugging. These approaches might be feasible when the set of languages that the systems support are semantically very similar. Such similarity, however, may be more rare in the future and is presently non-existent for parallel languages. Hence there has been a strong need to pursue other debugging methods that are capable of supporting a semantically diverse set of languages.

Maygen, the debugger generation system described in this thesis, is precisely such a debugging method. In light of the greater semantic diversity amongst programming languages, this system is more feasible than previous approaches to providing debugging support because of its ability to take into account different programming models. Additionally, generated debuggers exhibit a large degree of functional consistency, thus minimizing the user's overhead in learning a new debugging system for each new language.

The Maygen system provides for "quick and easy" creation of language-dependent debuggers for the respective target architectures. Such a feat is made possible by the system's imposition of interface protocols to be followed by language developers and architecture developers, provision of the glue necessary to not only smoothly connect the two interfaces but also serve as the core debugging engine, and provision of the framework that performs the actual gluing of the separate pieces.

Maygen has been shown to handle both a declarative language and an imperative language with reasonable results. The generated debuggers provided at least the minimal functionality needed for useful debugging without much additional effort on the part of language and architecture developers. Moreover, the generated debuggers were able to cater to the particular needs of each language and each architecture. Specifically, the generated OPAL debugger included several commands to provide for debugging features specific to Prolog-like languages, while the generated C debugger included commands to provide for debugging features specific to multiprocessor architectures.

Thus, the Maygen debugger generation system is a viable approach to providing debugging support for multiple languages, an increasingly important consideration as very different languages, such as parallel languages, are created.

7.2 Future Work

Because Maygen presents a feasible solution for providing debugging support, it is interesting to speculate upon what directions further research in the area of debugger generation might take.

Table 7.1: FUTURE MAYGEN WORK

Additional test sets
Improved test cases
Enhanced skeleton and additional interface routines

7.2.1 Maygen Prototype Enhancements

Several areas call for immediate improvement in the Maygen prototype. Most notably is the need to further explore the sample space of programming languages and their evaluation environments by creating additional test sets. A good third test set might be the Lisp[WH84, Bro86] language along with the Lisp runtime system. In addition, the existing test cases should be expanded where possible in order to produce debuggers with increased amounts of functionality. Finally, the existing debugger skeleton could be enhanced to provide a greater maximal amount of supported generated debugger functionality. This enhancement would most likely also require the specification of additional interface routines to be provided by the language and/or architecture developers. The suggested immediate modifications to the Maygen prototype are summarized in Table 7.1.

7.2.2 Related Areas to Explore

This section presents research areas suggested by Maygen work but of a much broader nature than that presented in the previous section. These areas can be grouped into four primary topics: creation of a Runtime System Interface (RSI); *characterization* of a language, architecture, or runtime system and the subsequent automatic generation of the respective Interface routines from each characterization; debugging of optimized code; and true debugging of parallel systems.

The division of the "world" that Maygen debuggers view is a rather unique one. Although the separation of a source program from that on which it runs, its evaluation environment, is a viable approach for the Maygen debugger, an alternative division might be to separate the source program from its runtime system as well as from its architecture.[1] This approach might provide for a "cleaner" and more traditional division; but, at the same time, this approach might be unnecessarily complex due to the desire to exploit higher-level abstractions when available, as described in Section 4.2.

A more thought-provoking area to explore is that of characterizing a source language in a way that a generation system could then automatically create the Source Language Interface routines defined in the Maygen system. Analogously, the characterizations of a machine architecture and of a runtime system, as well as the subsequent generation of Machine Architecture Interface and Runtime System Interface routines pose interesting questions. A key idea to keep in mind, though, is that although a method of characterization for these areas could prove theoretically interesting, it might not be practical in the context of efficient debugger generation. For example, language developers might find it much easier to conform to a set protocol for interface interaction (i.e., provide defined routines) rather than to conform to a "characterization method" for describing their language (i.e., provide a characterization of their language).

A third idea is that perhaps a debugger generation system could be developed that can better handle the debugging of optimized code. A start in that direction is that generated debuggers might be able to support semantically-unchanging optimizations—optimizations that are transparent to the user, such as dealing with register use versus memory use or caching. Another example of such an optimization would be one that moves a value to a storage place earlier than expected according to the source program, but that does not matter since that particular memory location is not needed any more. Hennessy examines the tradeoff between the optimization of code and the ability to symbolically debug it[Hen79], while Zellweger both studies the problem of debugging optimized programs and attempts to confront one aspect of this problem[Zel84].

A final area of research suggested by Maygen work is the generation of true parallel

[1]"Architecture" in this case refers to the evaluation environment minus the runtime system.

Table 7.2: DEBUGGER GENERATION SYSTEMS: AREAS TO EXPLORE

Separation of runtime system interface
Characterization of source languages
Generation of SLI routines
Characterization of machine architectures
Generation of MAI routines
Characterization of runtime systems
Generation of RSI routines
Handling of Optimized Code
Addition of True Parallelism

debuggers. Although Maygen's approach of having knowledge of multiple processing nodes but debugging only one node at a time is sufficient for this initial project in debugger generation, future work will probably need to better address parallel debugging issues.

The suggested areas to explore in further research of debugger generation systems are summarized in Table 7.2.

Without question, Maygen not only has presented an interesting and viable approach to providing debugging support for multiple language systems, but has also suggested a wealth of interesting research topics to pursue.

Appendix A

SLI Input File Template

```
%% INPUT FILE FORMAT FOR SOURCE LANGUAGE

SOURCE LANGUAGE NAME:
(e.g., CLU)
%###%
your_source_language_name

DEBUGGER LIBRARY PATH:
(e.g., /users/tsien/maygen/opal/)
%###%
your_debugger_library_path_name

DEBUGGER LIBRARY FILE NAME WITHOUT LEADING "lib" OR TRAILING ".a":
(e.g., for "libmf_debug.a", only use "mf_debug")
%###%
your_library_file_name

%%%%%%%%%%
%% Procedures: %%
%%%%%%%%%%
```

```
1. %###% Y
int init_sli(void)
  %%=====================================================
  %% Requires:  ---
  %% Modifies:  ---
  %% Effects:   Does any necessary initializations for SLI
  %%  Returns:  1 if everything initialized ok; 0 otherwise.
  %% Note:       (If procedure missing, assumed that there is
  %%           no initialization necessary for SLI)
  %%=====================================================

2. %###% [Y or N]
int map_proc_to_object(char *proc, char *label)
  %%=====================================================
  %% Requires:  proc is name user uses to refer to given procedure
  %%           label is name that compiler might use to refer to proc
  %% Modifies:  ---
  %% Effects:
  %%  Returns:  -1 if syntax error in proc spec
  %%           0 if procedure not found
  %%           n > 0, where n = object line corresponding to
  %%               the source code of proc
  %%=====================================================

3. %###% [Y or N]
int map_procbegin_to_object(char *proc, char *label)
  %%=====================================================
  %% Requires:  proc is name user uses to refer to given procedure
  %%           label is name that compiler might use to refer to proc
  %% Modifies:  ---
  %% Effects:
  %%  Returns:  -1 if syntax error in proc spec
  %%           0 if procedure not found
  %%           n > 0, where n = object line corresponding to
  %%               the beginning source line of proc
```

```
 %%===========================================================

4. %###% [Y or N]
int map_procend_to_object(char *proc, char *label)
 %%===========================================================
 %% Requires:  proc is name user uses to refer to given procedure
 %%            label is name that compiler might use to refer to proc
 %% Modifies:  ---
 %% Effects:
 %%  Returns:  -1 if syntax error in proc spec
 %%            0 if procedure not found
 %%            n, where n = object line corresponding to
 %%              the end source line of proc
 %%===========================================================

5. %###% [Y or N]
void trace_procedure(char *proc, char *label)
 %%===========================================================
 %% Requires:  proc is name user uses to refer to given procedure
 %%            label is name that compiler uses to refer to proc
 %% Modifies:  ---
 %% Effects:   Does whatever is necessary to trace proc
 %%  Returns:  ---
 %%===========================================================

6. %###% [Y or N]
int map_source_to_object(int srcline)
 %%===========================================================
 %% Requires:  srcline is an integer
 %% Modifies:  ---
 %% Effects:
 %%  Returns:  -1 if there is not source code at line srcline, or
 %%              if a breakpoint cannot be set at that line.
 %%            n, where n = object line corresponding to
 %%              line srcline.
 %%=========================================================

7. %###% Y
int read_symbols(char *filename)
 %%=========================================================
 %% Requires:  filename is the name of file with symbols to be read in
 %% Modifies:  ---
 %% Effects:   Loads user program symbols and/or labels;
 %%            sets global int program_start_loc to be address of
 %%            where program starts, if known.  Sets global
 %%            char user_program[] to be filename.
 %% Returns:   1 if symbols read successfully; 0 otherwise.
 %%=========================================================

8. %###% [Y or N]
void print_labels(char *arg1)
 %%=========================================================
 %% Requires:  arg1 is not required, but could be used
 %% Modifies:  ---
 %% Effects:   Prints out labels of user program currently loaded.
 %% Returns:   ---
 %%=========================================================

9. %###% [Y or N]
void list_procedures(char *arg1)
 %%=========================================================
 %% Requires:  arg1 is not required, but could be used
 %% Modifies:  ---
 %% Effects:   Prints out procedures of user program currently loaded.
 %% Returns:   ---
 %%=========================================================

10. %###% [Y or N]
void print_symbols(char *arg1)
 %%=========================================================
 %% Requires:  arg1 is not required, but could be used
 %% Modifies:  ---
 %% Effects:   Prints out symbols of user program currently loaded.
 %% Returns:  ---
 %%======================================================

11. %###% [Y or N]
void display_source_line_text(char *src_line)
 %%======================================================
 %% Requires:  src_line is a line of user program or is empty
 %% Modifies:  ---
 %% Effects:   Prints out source code corresponding to line src_line
 %%           of user program, or, if src_line is empty, then
 %%           shows current location in program and the source
 %%           code corresponding to current location.
 %% Returns:  ---
 %%======================================================

12. %###% [Y or N]
void untrace_procedure(char *proc, char *label)
 %%======================================================
 %% Requires:  proc is name user uses to refer to given procedure
 %%           label is name that compiler uses to refer to proc
 %% Modifies:  ---
 %% Effects:   Does whatever is necessary to untrace proc
 %% Returns:  ---
 %%======================================================

13. %###% [Y or N]
void print_sli_info(void)
 %%======================================================
 %% Requires:  ---
 %% Modifies:  ---
 %% Effects:   print source language information relevant to debugging
 %% Returns:  ---
```

```
  %%=====================================================================

14. %###% [Y or N]
int ProcessSLIArgs(int argc, char *argv[], char *progname)
  %%=====================================================================
  %% Requires:  progname is name of debugger program
  %% Modifies:  ---
  %% Effects:   Processes arguments, if any, of a generated debugger.
  %%            Prints a "Usage error:" line to output if returning 0.
  %% Returns:   1 if everything ok; 0 otherwise.
  %%=====================================================================

==================
EXTENSION COMMANDS
==================

NUMBER OF EXTENSION COMMANDS
(0 <= number <= 20)
%###%
<number>

For each extension command, specify:
(1) help line, including both name of command user will type
          and help string for help menu
          (e.g., "ta                    Toggle all-solutions.")
(2) invocation of name of routine to be called, using arguments
          arg1, arg2, arg3 (max 3 args)
          (e.g., "toggle_all_solutions();")
(3) extern reference line
          (e.g., "extern void toggle_all_solutions();")

EXAMPLE:

Extension Command 1
%###%
```

```
ta <n>                    Toggle all−solutions. n = max number of solns
%###%
toggle_all_solutions(arg1);
%###%
extern void toggle_all_solutions();
```

Appendix B

MAI Input File Template

```
%% INPUT FILE FORMAT FOR TARGET ARCHITECTURE

TARGET ARCHITECTURE NAME:
(e.g., CM5)
%###%
your_architecture_name

DEBUGGER LIBRARY PATH:
(e.g., /users/tsien/maygen/om/)
%###%
your_debugger_library_path_name

DEBUGGER LIBRARY FILE NAME WITHOUT LEADING "lib" OR TRAILING ".a":
(e.g., for "libmf_debug.a", only use "mf_debug")
%###%
your_library_file_name

ACTUAL NUMBER OF PROCESSING NODES IN TARGET ARCHITECTURE
("1" for a uniprocessor)
%###%
your_number
```

```
DESIRED NUMBER OF PROCESSING NODES IN TARGET ARCHITECTURE
(DESIRED NUMBER <= ACTUAL NUMBER: "1" for a uniprocessor)
%###%
your_number


%%%%%%%%%%%
%% Procedures: %%
%%%%%%%%%%%

1. %###% Y
int init_mai(void)
 %%=====================================================
 %% Requires:  ---
 %% Modifies:  ---
 %% Effects:   Does any necessary initializations for MAI
 %%  Returns:  1 if initialization successful; 0 otherwise.
 %%=====================================================

2. %###% Y
int program_loaded(void)
 %%=====================================================
 %% Requires:  ---
 %% Modifies:  ---
 %% Effects:
 %%  Returns:  1 if program is loaded
 %%            0 if program is not loaded
 %%=====================================================

3. %###% [Y or N]
int InstallMachineBreakpoint(int addr)
 %%=====================================================
 %% Requires:  addr is a valid code address of the current
 %%            program where a breakpoint can be set
 %% Modifies:  (object code)
 %% Effects:    Installs a breakpoint at addr such that when
 %%            program execution reaches addr, it halts
 %% Returns:  Original instruction (int) being replaced by breakpoint,
 %%            to be passed to UninstallMachineBreakpoint.  Returns
 %%            an integer < 0 if did not install correctly.
 %%======================================================

4. %###% Y
void continue_program(void)
 %%======================================================
 %% Requires:  ---
 %% Modifies:  ---
 %% Effects:   If program is running, continues running it.
 %%            Otherwise prints a message to user that program
 %%            should be started first.
 %% Returns:  ---
 %%======================================================

5. %###% [Y or N]
int UninstallMachineBreakpoint(int addr, int orig_instruction)
 %%======================================================
 %% Requires:  addr is a valid code address of the current
 %%            program where a breakpoint can be removed;
 %%            orig_instruction is identical to that returned by
 %%            InstallMachineBreakpoint
 %% Modifies:  (object code)
 %% Effects:    Uninstalls a breakpoint at addr such that when
 %%            program execution reaches addr, it no longer halts
 %%            due to this breakpoint.  Original instruction is
 %%            reinstated.
 %% Returns:  int n: n=0 if worked correctly; n<0 if did not work
 %%======================================================

6. %###% [Y or N]
int SetMachineProcBreakpoint(char *proc, int n, int trace_on)
```

```
  %%=========================================================
  %% Requires:  proc is name user uses to refer to a procedure or  which
  %%          a breakpoint is to be added
  %%          n is the code address where this procedure starts
  %% Modifies:  (object code)
  %% Effects:   Adds proc to list of procedure breakpoints by calling
  %%          int add_to_proc_breakpt_list(char *proc, int addr,
  %%          int trace_on).  (1 if good; 0 if bad)
  %%          (in SKEL) and adds corresponding machine address
  %%          breakpoint(s) from list by calling (in SKEL:)
  %%          int add_to_machine_breakpt_list(int addr).
  %%          (1 if good; 0 if bad)
  %%  Returns:  1 if set successfully; 0 otherwise
  %%=========================================================

7. %###% [Y or N]
int ClearMachineProcBreakpoint(char *proc, int n)
  %%=========================================================
  %% Requires:  proc is name user uses to refer to a procedure on which
  %%          there is a breakpoint to be removed.
  %%          n is the code address where the procedure starts
  %% Modifies:  (object code)
  %% Effects:   Removes proc from list of procedure breakpoints by calling
  %%          int remove_from_proc_breakpt_list(char *proc, int addr)
  %%          (1 if good; 0 if bad returned)
  %%          (in SKEL) and removes corresponding machine address
  %%          breakpoints from list by calling (in SKEL:)
  %%          int remove_from_machine_breakpt_list(int addr)
  %%          (1 if good; 0 if bad returned)
  %%  Returns:  1 if successful; 0 otherwise
  %%=========================================================

8. %###% Y
int read_program(char *filename)
  %%=========================================================
  %% Requires:  filename is the name of file to be read in
  %% Modifies:  (machine state)
  %% Effects:   Loads user program; loads the code into the code
  %%            memory.  Set flags such that program_loaded() will
  %%            return true.  Reinitialize memory, etc.
  %%  Returns:  1 if program read successfully, 0 otherwise
  %%=====================================================
```

9. %###% [Y or N]

```
void print_register_contents(char *arg1, char *arg2)
  %%=====================================================
  %% Requires:  arg1 is possibly an environment
  %% Modifies:  ---
  %% Effects:   Prints the contents of the machine registers;
  %%            If env is given, only prints that environment
  %%  Returns:  ---
  %%=====================================================
```

10. %###% Y

```
void run_program(char *a1)
  %%=====================================================
  %% Requires:  arg1 is empty or is a line number at which to begin
  %%            execution
  %% Modifies:  ---
  %% Effects:   Reports that user program is already running (and
  %%            suspended) or else begins to run the program.
  %%  Returns:  ---
  %%=====================================================
```

11. %###% [Y or N]

```
void do_step(char *arg1, char *arg2)
  %%=====================================================
  %% Requires:  arg1 is empty or the number of steps user wants to step.
  %%            arg2 is empty or the location from which to begin stepping
  %% Modifies:  ---
```

```
 %% Effects:   Executes arg1 steps of user program, beginning at
 %%           location arg2.
 %% Returns:  ---
 %%========================================================

12. %###% [Y or N]
void do_big_step(char *arg1)
 %%========================================================
 %% Requires:  arg1 is empty or the location from which to begin stepping
 %% Modifies:  ---
 %% Effects:   Executes a process/procedure of user program, beginning at
 %%           location arg1.
 %% Returns:  ---
 %%========================================================

13. %###% Y
void reset_machine(void)
 %%========================================================
 %% Requires:  ---
 %% Modifies:  machine state
 %% Effects:   Resets the machine state, sets running to false (0)
 %%========================================================

14. %###% [Y or N]
void print_mai_info(void)
 %%========================================================
 %% Requires:  ---
 %% Modifies:  ---
 %% Effects:   Prints out information about user program, debugger
 %%           status, etc.
 %% Returns:  ---
 %%========================================================

15. %###% [Y or N]
int ProcessMAIArgs(int argc, char *argv[], char *progname)
```

```
  %%=====================================================
  %% Requires:  progname is name of debugger program
  %% Modifies:  ---
  %% Effects:   Processes arguments, if any, of a generated debugger.
  %%            Prints a "Usage error:" line to output if returning 0.
  %%  Returns:  1 if everything ok; 0 otherwise.
  %%=====================================================

16. %###% [Y or N]
int change_node(int arg1)
  %%=====================================================
  %% Requires:  arg1 is an integer specifying the new node to be
  %%            debugged.  Is already checked for <= max available
  %%            and > 0
  %% Modifies:  machine state
  %% Effects:   Does the necessary internal state changes to debug
  %%            node number arg1
  %%  Returns:  1 if everything ok; 0 otherwise.
  %%=====================================================

17. %###% [Y or N]
int resize_number_nodes(int arg1)
  %%=====================================================
  %% Requires:  arg1 is an integer specifying the new desired number
  %%            of processing nodes.  Is already checked for <= max
  %%            and > 0
  %% Modifies:  machine state
  %% Effects:   Does the necessary internal state changes to alter
  %%            desired number of nodes available to arg1
  %%  Returns:  1 if everything ok; 0 otherwise.
  %%=====================================================

================
EXTENSION COMMANDS
================
NUMBER OF EXTENSION COMMANDS
(0 <= number <= 20)
%###%
<number>

For each extension command, specify:
(1) help line, including both name of command user will type
        and help string for help menu
        (e.g., "ta                    Toggle all-solutions.")
(2) invocation of name of routine to be called, using arguments
        arg1, arg2, arg3 (max 3 args)
        (e.g., "toggle_all_solutions();")
(3) extern reference line
        (e.g., "extern void toggle_all_solutions();")

EXAMPLE:

Extension Command 1
%###%
ta <n>                  Toggle all-solutions. n = max number of solns
%###%
toggle_all_solutions(arg1);
%###%
extern void toggle_all_solutions();
```

Appendix C

Sample OM Virtual Machine MAI Input File

```
%% MAI INPUT FILE FOR TARGET ARCHITECTURE OM VIRTUAL MACHINE

TARGET ARCHITECTURE NAME:
%###%
OM

DEBUGGER LIBRARY PATH:
%###%
/users/tsien/maygen/om/

DEBUGGER LIBRARY FILE NAME WITHOUT LEADING "lib" OR TRAILING ".a":
%###%
mg_mai

ACTUAL NUMBER OF PROCESSING NODES IN TARGET ARCHITECTURE
%###%
1
```

```
DESIRED NUMBER OF PROCESSING NODES IN TARGET ARCHITECTURE
%###%
1


%%%%%%%%%%%
%% Procedures: %%
%%%%%%%%%%%
1. %###% Y
int init_mai(void)

2. %###% Y
int program_loaded(void)

3. %###% Y
int InstallMachineBreakpoint(int addr)

4. %###% Y
void continue_program(void)

5. %###% Y
int UninstallMachineBreakpoint(int addr, int orig_instruction)

6. %###% Y
int SetMachineProcBreakpoint(char *proc, int n, int trace_on)

7. %###% Y
int ClearMachineProcBreakpoint(char *proc, int n)

8. %###% Y
int read_program(char *filename)

9. %###% Y
void print_register_contents(char *arg1, char *arg2)

10. %###% Y
```

```
void run_program(char *a1)

11. %###% Y
void do_step(char *arg1, char *arg2)

12. %###% Y
void do_big_step(char *arg1)

13. %###% Y
void reset_machine(void)

14. %###% Y
void print_mai_info(void)

15. %###% Y
int ProcessMAIArgs(int argc, char *argv[], char *progname)

16. %###% N
int change_node(int arg1)

17. %###% N
int resize_number_nodes(int arg1)

==================
EXTENSION COMMANDS
==================
NUMBER OF EXTENSION COMMANDS
%###%
12

Extension Command 1
%###%
ta                        Toggle all-solutions.
%###%
toggle_all_solutions();
```

```
%###%
extern void toggle_all_solutions();


Extension Command 2
%###%
tb                          Toggle breadth-first search.
%###%
toggle_breadth_first();
%###%
extern void toggle_breadth_first();


Extension Command 3
%###%
tq                          Toggle quiet mode.
%###%
toggle_quiet_mode();
%###%
extern void toggle_quiet_mode();


Extension Command 4
%###%
tp                          Toggle process trace.
%###%
toggle_process_trace();
%###%
extern void toggle_process_trace();


Extension Command 5
%###%
ti                          Toggle instruction trace.
%###%
```

toggle_instruction_trace();

%###%

extern void toggle_instruction_trace();

Extension Command 6

%###%

td Toggle symbolic reg display.

%###%

toggle_symbolic_display();

%###%

extern void toggle_symbolic_display();

Extension Command 7

%###%

pc Print code from <n> to <m>.

%###%

print_code(arg1, arg2);

%###%

extern void print_code();

Extension Command 8

%###%

ph Print heap from <n> to <m>.

%###%

print_heap(arg1, arg2);

%###%

extern void print_heap();

Extension Command 9

%###%

pm Print message (detailed contents of M reg).

%###%

print_message_info(arg1, arg2);

%###%

extern void print_message_info( );

Extension Command 10

%###%

pp   Print process (detailed contents of P reg).

%###%

print_process_info(arg1, arg2);

%###%

extern void print_process_info( );

Extension Command 11

%###%

pq   Print message queue.

%###%

print_queue_contents( );

%###%

extern void print_queue_contents( );

Extension Command 12

%###%

pt   Print process tree.

%###%

print_process_tree( );

%###%

extern void print_process_tree( );

Bibliography

[ASU86] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers. Principles, Techniques, and Tools.* Addison-Wesley Publishing Company, 1986.

[BBK$^+$82] James Bodwin, Laurette Bradley, Kohji Kanda, Diane Litle, and Uwe Pleban. Experience with an Experimental Compiler Generator Based on Denotational Semantics. In *Proceedings of the ACM SIGPLAN Symposium on Compiler Construction, published in ACM SIGPLAN Notices*, pages 216–223. University of Michigan, June 1982. volume 17, number 6.

[Bea83] Bert Beander. VAX DEBUG: An Interactive, Symbolic, Multilingual Debugger. *ACM Sigplan Notices*, 18(8):173–179, August 1983. Proceedings of the ACM Sigsoft/Sigplan Software Engineering Symposiun on High Level Debugging.

[Bir82] Peter L. Bird. An Implementation of a Code Generator Specification Language for Table Driven Code Generators. In *Proceedings of the ACM SIGPLAN Symposium on Compiler Construction, published in ACM SIGPLAN Notices*, pages 44–55. University of Michigan, June 1982. volume 17, number 6.

[Bro79] P. J. Brown. *Writing Interactive Compilers and Interpreters.* Wiley Series in Computing. John Wiley & Sons, Computing Laboratory, University of Kent at Canterbury, 1979.

[Bro86] Hank Bromley. *Lisp Lore: A Guide To Programming the Lisp Machine.* Kluwer Academic Publishers, 1986.

[Car83] James R. Cardell. Multilingual Debugging with the SWAT High-level Debugger. *ACM Sigplan Notices*, 18(8):180–189, August 1983. Proceedings of the ACM Sigsoft/Sigplan Software Engineering Symposium on High Level Debugging.

[Con90] John S. Conery. Parallel Logic Programs on the HP Mayfly. Technical Report CIS-TR-90-22, University of Oregon, December 7 1990.

[Con91] John S. Conery. *OPAL User's Guide.* University of Oregon, February 6 1991.

[Con92] John S. Conery. Parallel Logic Programs on the Mayfly. *Lisp and Symbolic Computation: An International Journal*, 5(1/2):49–72, May 1992.

[Dav92] Al Davis. Mayfly: A General-Purpose, Scalable, Parallel Processing Architecture. *Lisp and Symbolic Computation: An International Journal*, 5(1/2):7–47, May 1992.

[DNF79] Michael K. Donegan, Robert E. Noonan, and Stefan Feyock. A Code Generator Generator Language. In *Proceedings of the ACM SIGPLAN Symposium on Compiler Construction, published in ACM SIGPLAN Notices*, pages 58–64. College of William and Mary, August 1979. volume 14, number 8.

[Edw75] Edwin Satterthwaite Jr. *Source Language Debugging Tools.* PhD thesis, Stanford University, May 1975. Outstanding Dissertations in the Computer Sciences. Garland Publishing, Inc. 1979.

[FJ88] Charles N. Fischer and Richard J. LeBlanc Jr., editors. *Crafting a Compiler.* The Benjamin/Cummings Publishing Company, Inc., 2727 Sand Hill Road, Menlo Park, CA 94025, 1988.

[GG78] R. S. Glanville and S. L. Graham. A New Method for Compiler Code Generation. In *5th ACM Symposium on Principles of Programming Languages*, 1978.

[Hen79] John L. Hennessy. Symbolic Debugging of Optimized Code. Technical Report 175, Stanford University, Computer Systems Laboratory, July 1979.

[Joh75] S. C. Johnson. YACC: Yet Another Compiler-Compiler. Computing Science Technical Report 32, Bell Laboratories, Murray Hill, NJ, 1975.

[Joh77] Mark Scott Johnson. The Design of a High-Level, Language-Independent Symbolic Debugging System. In *Proceedings of the Annual Conference of the ACM*, pages 315–322, 2075 Wesbrook Mall, Vancouver, British Columbia V6T 1W5, 1977. University of British Columbia.

[Joh78] Mark Scott Johnson. *The Design and Implementation of a Run-Time Analysis and Interactive Debugging Environment.* PhD thesis, University of British Columbia, August 1978. Technical Report 78-6. 148pp.

[Joh81] Mark Scott Johnson. Dispel: A Run-time Debugging Language. *Computer Languages*, 6(2):79–94, 1981.

[Joh82] Mark Scott Johnson. A Software Debugging Glossary. *ACM Sigplan Notices*, 17:53–70, February 1982.

[Kac90] Peter Kacsuk. *Execution Models of Prolog for Parallel Computers.* Research Monographs in Parallel and Distributed Computing. The MIT Press, 1990.

[KR88] Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language, Second Edition.* Prentice Hall Software Series. Prentice Hall, 1988.

[KW91] Stephen G. Kochan and Patrick H. Wood. *Topics in C Programming.* John Wiley & Sons, Inc, 1991.

[LJG82] Rudolf Landwehr, Hans-Stephan Jansohn, and Gerhard Goos. Experience with an Automatic Code Generator Generator. In *Proceedings of the ACM SIGPLAN Symposium on Compiler Construction, published in ACM SIGPLAN Notices*, pages 56–66. Universitat Karlsruhe, Institut fur Informatik II, June 1982. volume 17, number 6.

[Mak91] Ronald Mak. *Writing Compilers & Interpreters.* John Wiley & Sons, Inc, 1991.

[MKR79] D R. Milton, L. W. Kirchhoff, and B. R. Rowland. An ALL(1) Compiler Generator. In *Proceedings of the ACM SIGPLAN Symposium on Compiler Construction, published in ACM SIGPLAN Notices*, pages 152–157, Naperville, IL 60540, August 1979. Bell Laboratories. volume 14, number 8.

[MM83] Henry McGilton and Rachel Morgan. *Introducing the UNIX System.* McGraw-Hill Software Series For Computer Professionals. McGraw-Hill Book Company, 1983.

[Ras82] Martin R. Raskovsky. Denotational Semantics as a Specification of Code Generators. In *Proceedings of the ACM SIGPLAN Symposium on Compiler Construction, published in ACM SIGPLAN Notices*, pages 230–244. Essex University, June 1982. volume 17, number 6.

[Sch88] David A. Schmidt, editor. *Denotational Semantics. A Methodology for Language Development.* Wm. C. Brown Publishers, Dubuque, Iowa, 1988.

[Sto77] Joseph E. Stoy, editor. *Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory.* The MIT Press Series in Computer Science. The MIT Press, Cambridge, Massachusetts, 1977. Foreword by Dana S. Scott.

[SW90] A. Schleiermacher and J. F. H. Winkler. The Implementation of ProTest: A Prolog Debugger for a Refined Box Model. *Software—Practice and Experience*, 20(10):985–1006, October 1990.

[Tof90] Mads Tofte. *Compiler Generators. What They Can Do, What They Might Do, and What They Will Probably Never Do*, volume 19 of *EATCS Monographs on Theoretical Computer Science.* Springer-Verlag, 1990. Editors: W. Brauer and G. Rozenberg and A. Salomaa.

[Tro82] James Henry Troisi. *An Interpreter and Symbolic Debugger for C.* PhD thesis, Massachusetts Institute of Technology, August 1982.

[War83] D. H. D. Warren. An Abstract Prolog Instruction Set. Technical Note 309, SRI International, October 1983.

[WH84] Patrick Winston and Berthold Klaus Paul Horn. *Lisp, Second Edition.* Addison-Wesley Publishing Company, 1984.

[Zel84] Polle Trescott Zellweger. *Interactive Source-Level Debugging of Optimized Programs.* PhD thesis, University of California, Berkeley, May 1984. Xerox PARC CSL-84-5.